Procesy i potoki

dr inż. Sławomir Samolej
Katedra Informatyki i Automatyki
Politechnika Rzeszowska

Program przedmiotu oparto w części na materiałach opublikowanych na:
http://wazniak.mimuw.edu.pl/
oraz
Na materiałach opracowanych przez
dr inż. Jędrzeja Ułasiewicza:
jedrzej.ulasiewicz.staff.iiar.pwr.wroc.pl

Plan zajęć

- Wykład
  - Wprowadzenie do zagadnień współbieżności
  - Programowanie wieloprocesowe i wielowątkowe
  - Rozwiązania klasycznych problemów współbieżności: producent-konsument, czytelnicy i pisarze, 5 ucztujących filozofów
  - Programowanie aplikacji rozproszonych
  - Klastry komputerowe (dr inż. Tomasz Rak)
  - Programowanie aplikacji współbieżnych i rozproszonych w Java, obliczenia w Grid i Cloud (dr inż. Wojciech Rząsa)
- Laboratorium
  - Analiza i programowanie wybranych aplikacji współbieżnych i rozproszonych

Zasady zaliczenia

- Uzyskanie pozytywnej oceny z laboratorium (sprawdziany)
- Zdanie egzaminu

LITERATURA

- http://wazniak.mimuw.edu.pl/
- **Linux : programowanie / Neil Matthew, Richard Stones, RM, 1999**
- **Linux. Niezbędnik programisty/ John Fusco, Helion 2009**
- **Zaawansowane programowanie w systemie Linux / Neil Matthew, Richard Stones, Helion, 2002**
- **Systemy czasu rzeczywistego QNX6 Neutrino / Jędrzej Ułasiewicz, Wydawnictwo btc, 2007.**
- Programowanie w Linuksie / K. Kuźniar, K. Lal, T. Rak, Helion, 2012
- Podstawy programowania współbieżnego i rozproszonego / Mordechai Ben-Ari, WNT, 2009
- Java 2. Techniki zaawansowane. Wydanie II /Cay Horstmann, Gary Cornell, Helion, 2005
- Modele i metody inżynierii oprogramowania systemów czasu rzeczywistego/ T. Szmuc, Uczelniane Wydawnictwa Naukowo-Dydaktyczne AGH, Kraków 2001

Co nas będzie interesowało?

- Program współbieżny składa się z kilku (co najmniej dwóch) współbieżnych procesów sekwencyjnych, które muszą się ze sobą komunikować lub synchronizować swoje działania.
- Nie interesują nas procesy rozłączne, czyli takie które działają niezależnie od siebie, nie wymagając żadnych działań synchronizacyjnych ani nie wymieniając między sobą danych.
- Zajmiemy się omówieniem mechanizmów udostępnianych przez systemy operacyjne do synchronizacji procesów. Zwrócimy uwagę na pułapki, w jakie może wpaść programista.

Synchronizacja i komunikacja

- Poprawne zachowanie programu współbieżnego zależy od synchronizacji i komunikacji pomiędzy procesami
- **Synchronizacja** to wypełnienie ograniczeń dotyczących kolejności wykonywania pewnych akcji przez procesy (np. pewna akcja wykonywana przez jeden proces może nastąpić tylko wtedy, gdy pewna inna akcja została wykonana w innym procesie)
- **Komunikacja** to przekazywanie informacji od jednego procesu do innego
- Te dwa pojęcia są ze sobą „splecione” ponieważ komunikacja nie może się odbyć bez synchronizacji, a synchronizacja może być potraktowana jako komunikacja bez wymiany danych
- Wymiana danych jest zwykle oparta na **współdzieleniu zmiennych** albo **przekazywaniu wiadomości**

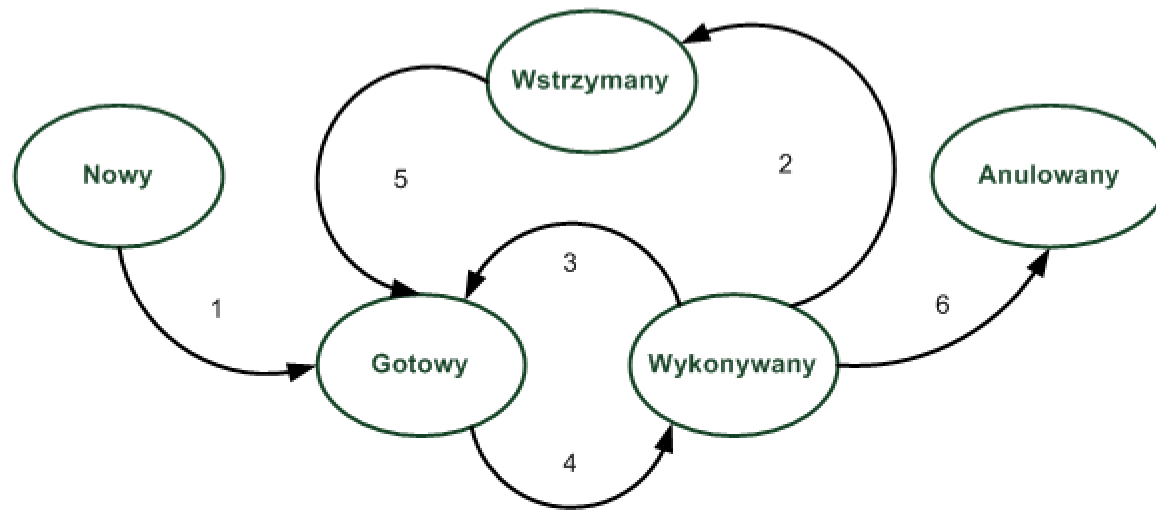
Proces

- **Proces** to przestrzeń adresowa i pojedynczy wątek sterujący, który działa w tej przestrzeni, oraz potrzebne do tego zasoby systemowe
- Minimalne zasoby do wykonywania się procesu to:
  - Procesor
  - Pamięć
  - Urządzenia wejścia-wyjścia
- W systemach wielozadaniowych w istocie każda instancja działającego programu jest procesem.

Przypomnienie – uruchomienie procesu

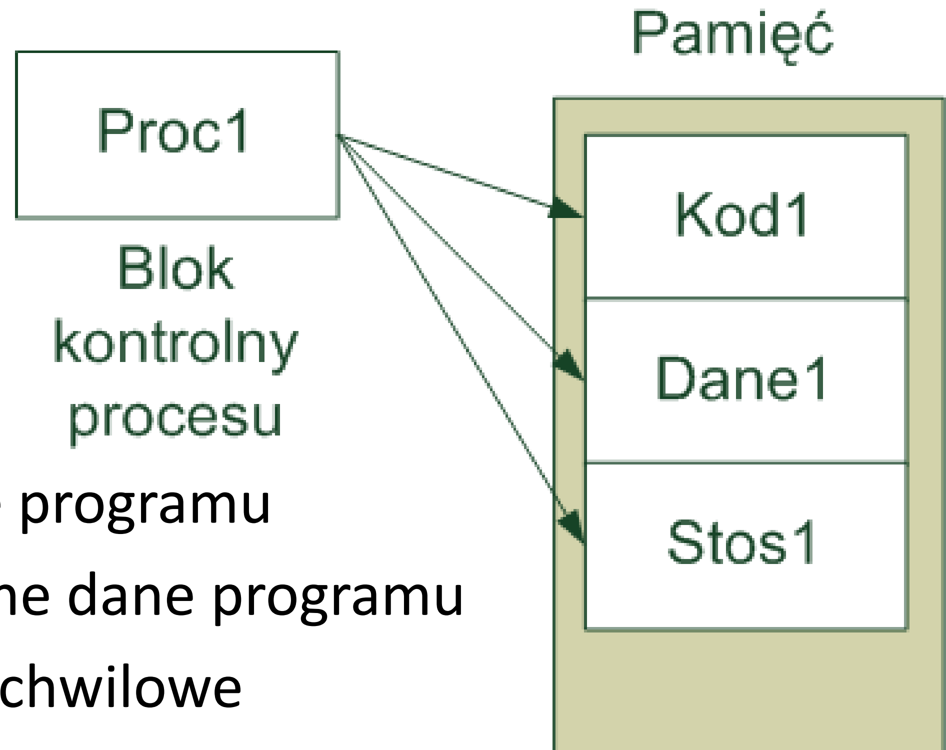
- Podczas uruchamiania programu w systemie operacyjnym następuje przeniesienie kodu programu z jakiegoś nośnika (najczęściej systemu plików) do pamięci operacyjnej.
- Kod programu jest „otaczany” pewnymi strukturami danych identyfikującymi go na czas wykonywania i zapewniającymi jego ochronę (tzw. blok kontrolny procesu).
- Następuje także powiązanie (planowanie przydziału) z programem zasobów systemu mikroprocesorowego, z których będzie korzystał.
- Zasobami przyszłego procesu są czas procesora, pamięć, system plików oraz urządzenia wejścia-wyjścia. Tak przygotowany do wykonywania program staje się **procesem**, który jest gotowy do wykonywania.

Przypomnienie – podstawowe stany procesu



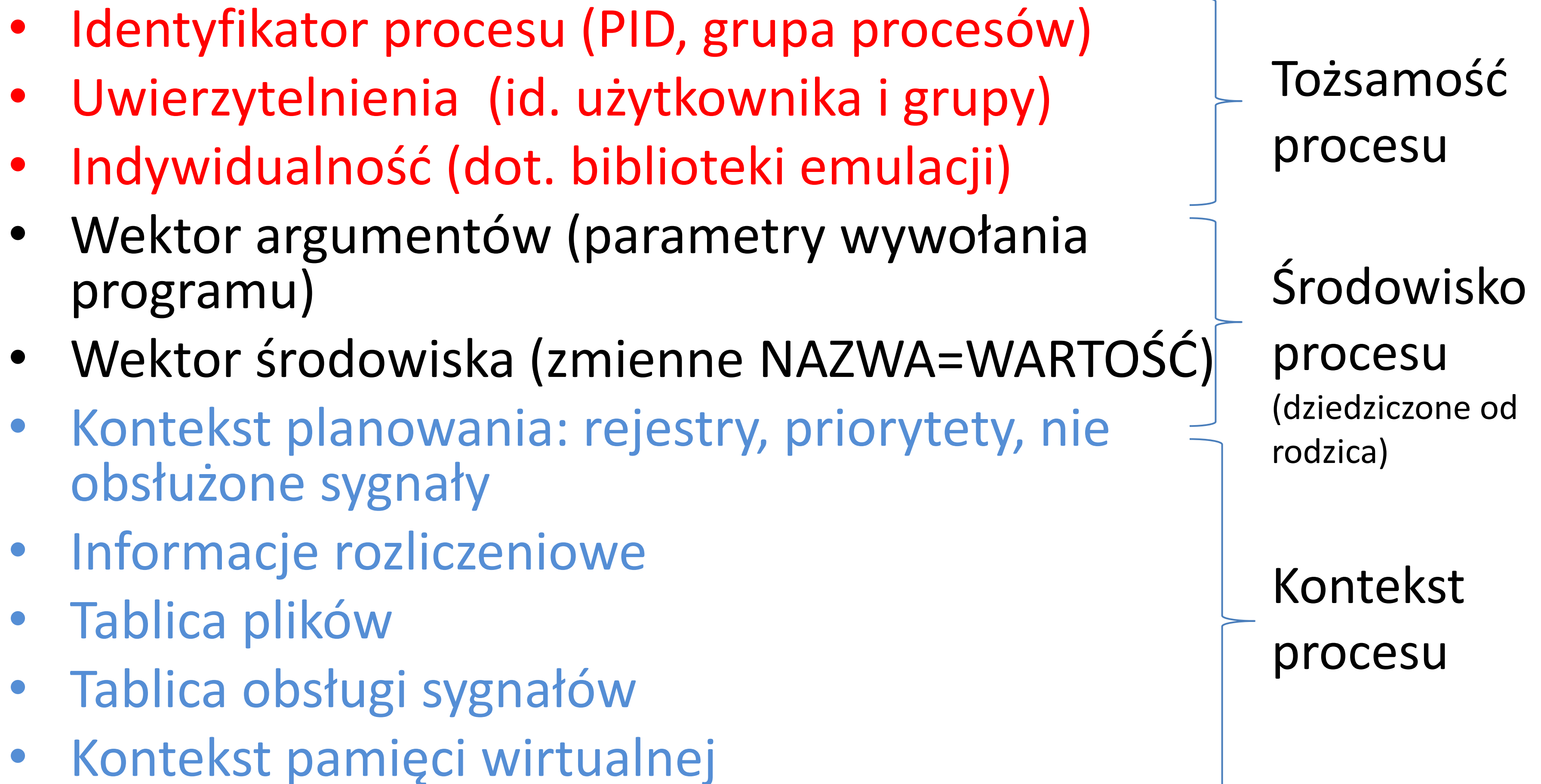
1. Utworzenie procesu
2. Proces żąda zasobu, który nie jest dostępny
3. Wystąpiło przerwanie (wywłaszczenie) lub proces zwolnił procesor dobrowolnie
4. Proces został wybrany do wykonywania
5. Potrzebny zasób został zwolniony (przerwanie wej/wyj lub inicjatywa bieżącego procesu)
6. Zakończenie procesu

Przypomnienie – struktury danych powiązane z procesem



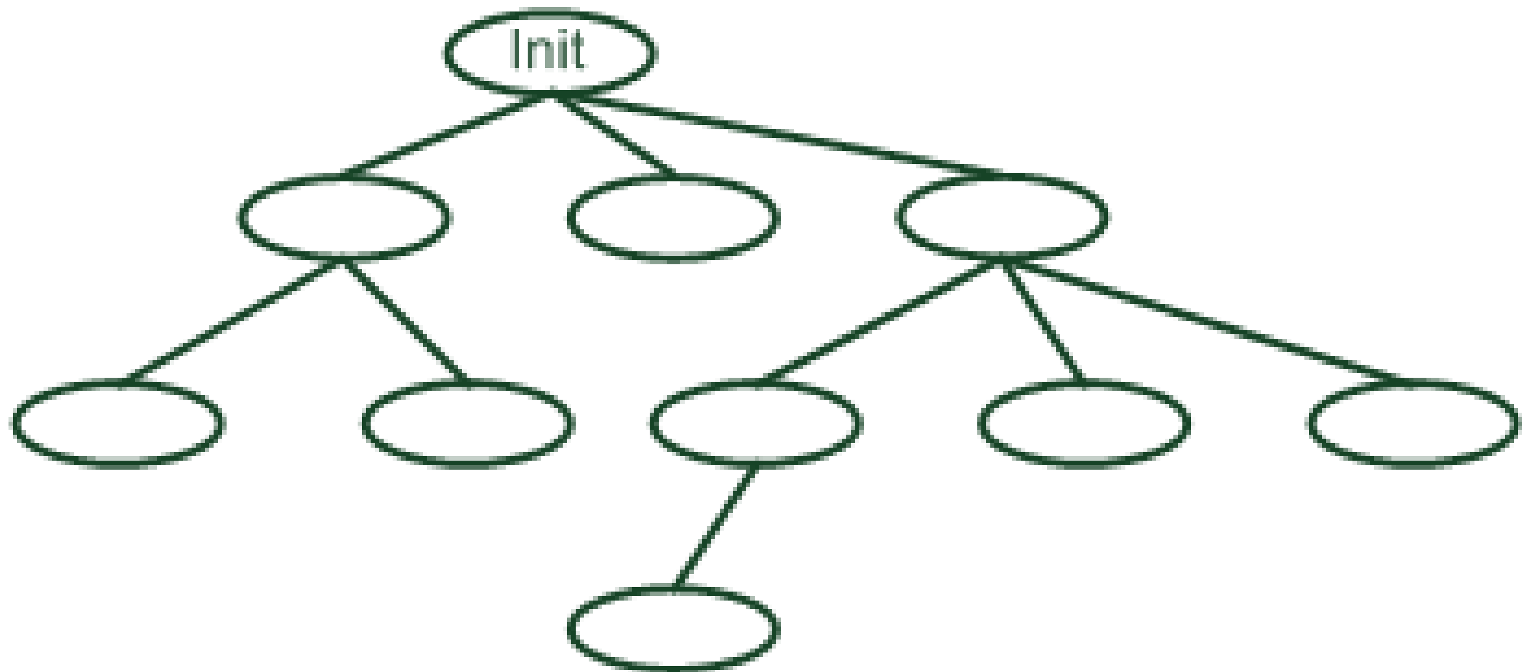
- Segment kodu – instrukcje programu
- Segment danych – statyczne dane programu
- Segment stosu – zmienne chwilowe
- Segment sterty – zmienne chwilowe jawnie alokowane i zwalniane przez programistę
- Blok kontrolny procesu – dane które o procesie przechowuje system operacyjny

Przypomnienie – blok kontrolny procesu



Drzewo procesów

- Procesy tworzą drzewo
- Każdy proces ma dokładnie jeden proces – rodzic
- Procesy mogą posiadać wiele procesów-dzieci



Polecenia Linux zarządzające procesami

- **ps** – pobieranie informacji o aktywnych procesach
  ps –ef – wszystkie procesy w systemie
  ps – axjf – drzewo procesów
  ps – sLf – informacja o wątkach
- **top** – dynamiczna lista aktywnych procesów w systemie
- **pstree** – drzewo procesów w systemie
- **monitor systemu**
- **nice** – uruchomienie programu z zadanym priorytetem
- **renice** – zmiana priorytetu uruchomionego procesu
- **kill** – przesłanie sygnału do procesu pracującego w systemie operacyjnym

Interpretacja kolumn polecenia PS

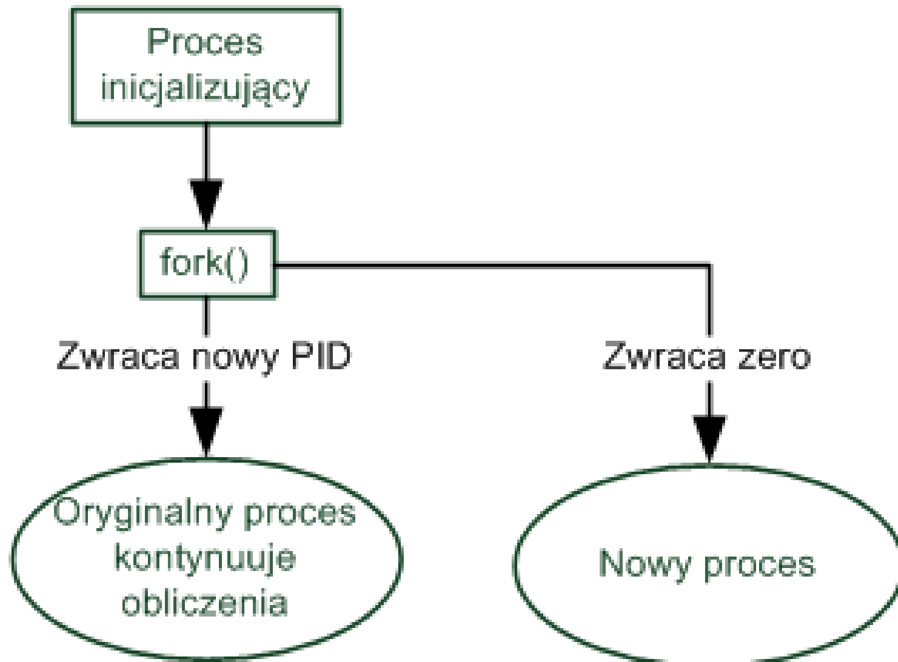
- UID – identyfikator użytkownika procesu
- PID – numer procesu w systemie
- PPID – numer rodzica procesu
- C- stopień wykorzystania procesora
- STIME – czas uruchomienia
- TTY – konsola, z której uruchomiono proces
- TIME – wykorzystany do tej pory czas procesora
- COMMAND – polecenie, które uruchomiło proces
- STAT – status: S-sleep, R-running,

Funkcje informujące i zarządzające procesami

Funkcja	Opis
pid_t getpid(void);	Zwróć swój identyfikator procesu
pid_t getppid(void);	Zwróć identyfikator procesu macierzystego
unsigned int sleep(unsigned int seconds);	„Uśpij” bieżący wątek/proces na zadaną ilość sekund
int pause(void);	Zatrzymaj wykonywanie programu do momentu, gdy otrzyma on sygnał, który go „zabije” lub który zostanie obsłużony

Duplikowanie procesu (1)

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```




```
pid_t new_pid;
new_pid = fork();
switch(new_pid) {
case -1 : /* Error */
break;
case 0 : /* We are child */
break;
default : /* We are parent */
break;
}
```

Duplikowanie procesu (2)

- Wywołanie systemowe tworzy nowy proces potomny, który jest identyczny z procesem wywołującym, ale posiada unikatowy PID, a jego PPID jest ustawiany na proces wywołujący.
- Nowy proces wywołujący dziedziczy po procesie macierzystym
  - Przestrzeń danych (zmienne)
  - Otwarte deskryptory i strumienie katalogowe
- Wywołanie fork w procesie macierzystym zwraca PID nowego procesu potomnego
- Nowy proces pracuje dokładnie tak samo, jak oryginał, ale w procesie potomnym fork zwraca zero (dzięki temu procesy mogą rozróżnić, który jest który)

Duplikowanie procesu – przykładowa aplikacja

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{       pid_t pid; int n;
        char *message;
        printf("fork program starting\n");
        pid = fork();
        switch(pid)
        {       case -1:        perror("fork failed");
                                exit(1);
                case 0:         message = "This is the child";
                                n = 5;
                                break;
                default:        message = "This is the parent";
                                n = 3;
                                break;
}       for(; n > 0; n--) {
        puts(message); sleep(1);
        }
        exit(0);}
```

Rezultat działania programu

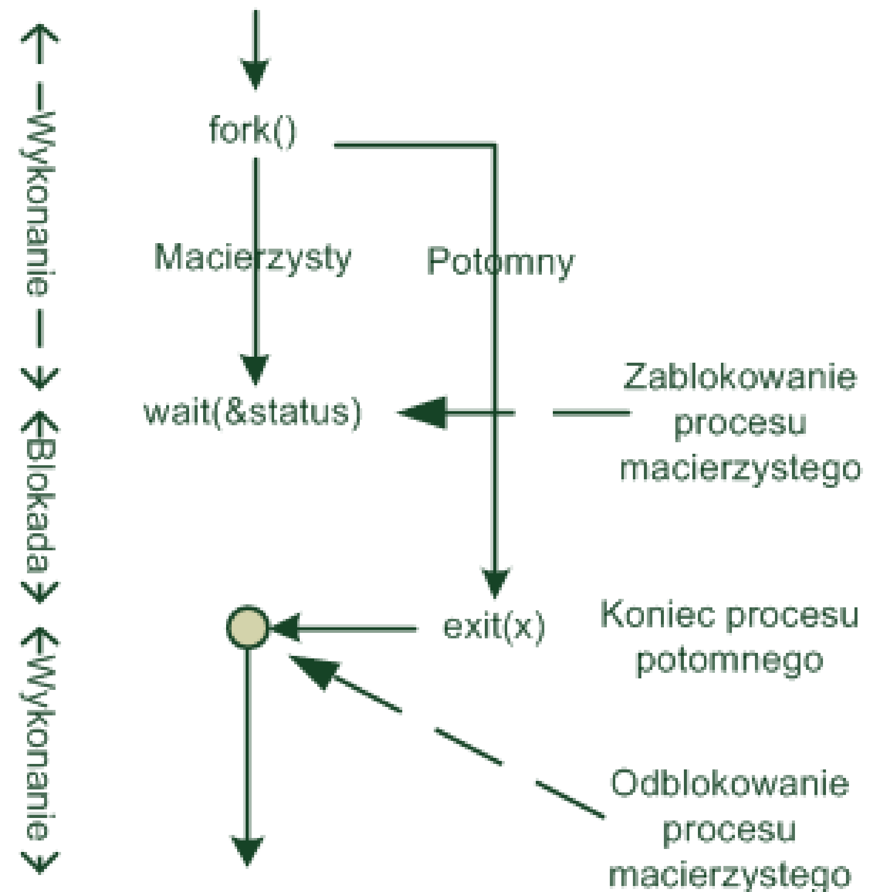
```
$ ./fork1
fork program starting
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the child
$ This is the child
This is the child
```

- Program działa jako 2 procesy
- Proces potomny wysyła komunikat 5 razy, macierzysty -3
- Proces macierzysty kończy działanie przed zakończeniem procesu dziecka
- W komunikaty wmieszał się znak powłoki…

Oczekiwanie na proces (1)

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *stat_val);
```

- Gdy proces potomny nie zakończył się funkcja wait powoduje zablokowanie procesu macierzystego aż do zakończenia się procesu potomnego. Gdy ten się zakończy zwracany jest jego PID oraz status
- Gdy proces potomny zakończył się zanim wykonano funkcję wait nie występuje blokada procesu macierzystego. Funkcja zwraca PID zakończonego procesu oraz jego status
- Gdy brak jakichkolwiek procesów potomnych funkcja wait zwraca –1.



Odczytywanie statusu przechwyconego przez funkcję wait

Makro	Definicja
WIFEXITED(stat_val)	Niezerowe, jeśli proces potomny normalnie zakończył pracę
WEXITSTATUS(stat_val)	Jeśli WIFEXITED jest niezerowe, zwraca kod wyjściowy procesu potomnego
WIFSIGNALED(stat_val)	Niezerowe, jeśli proces potomny zakończył pracę po otrzymaniu nie przechwyconego sygnału
WTERMSIG(stat_val)	Jeśli WIFSIGNALED jest niezerowe, zwraca numer sygnału
WIFSTOPPED(stat_val)	Niezerowe, jeśli proces potomny zatrzymał się po otrzymaniu sygnału
WSTOPSIG(stat_val)	Jeśli WIFSTOPPED jest niezerowe, zwraca numer sygnału

Zastosowanie wait – przykładowa aplikacja

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
int main()
{ pid_t pid;  char *message;  int n;
  int exit_code;
  printf("fork program starting\n");
  pid = fork();
 switch(pid)
{ case -1:  perror("fork failed");
            exit(1);
 case 0:    message = "This is the child";
            n = 5;
            exit_code = 37;
            break;
default:    message = "This is the parent";
            n = 3;
            exit_code = 0;
            break;
}
```

```
for(; n > 0; n--) {  puts(message);
                     sleep(1);
}

if (pid != 0) {
int stat_val;
pid_t child_pid;
child_pid = wait(&stat_val);
printf("Child has finished: PID = %d\n",
child_pid);
if(WIFEXITED(stat_val))
printf("Child exited with code %d\n",
WEXITSTATUS(stat_val));
else
printf("Child terminated abnormally\n");
}
exit(exit_code);
}
```

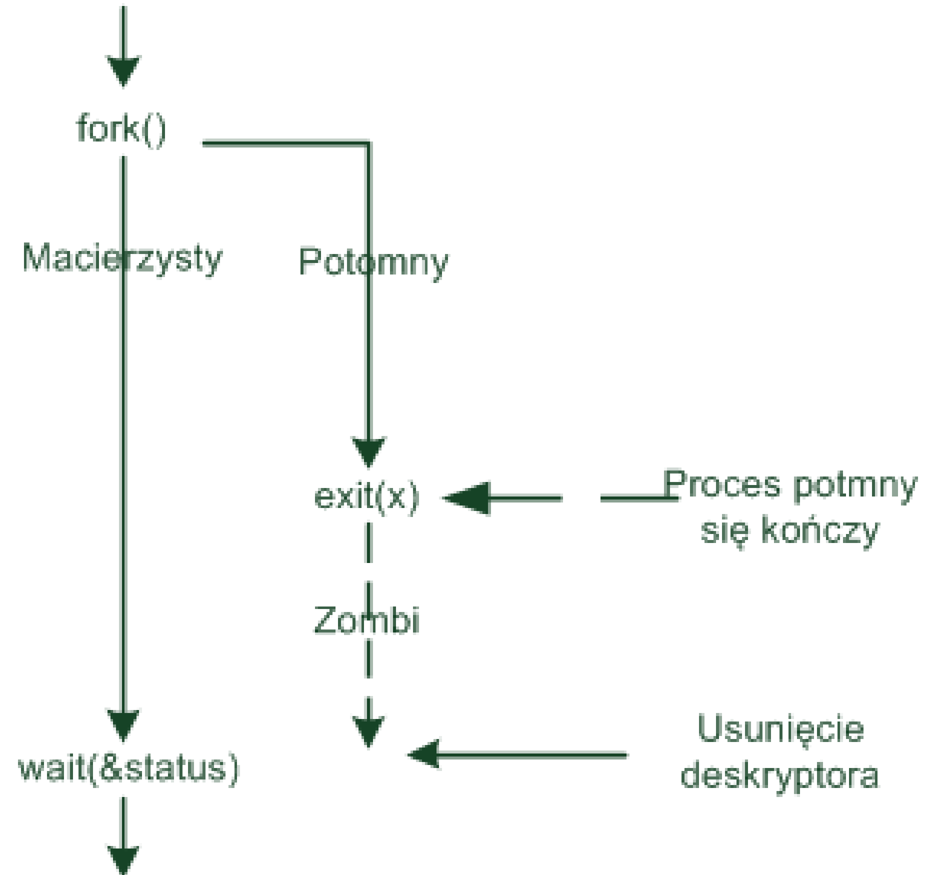
Rezultat programu

```
$ ./wait
fork program starting
This is the child
This is the parent
This is the parent
This is the child
This is the parent
This is the child
This is the child
This is the child
Child has finished: PID = 1582
Child exited with code 37
$
```

- Proces macierzysty korzysta z wywołania wait i zawiesza swoje działanie
- Kiedy zostaje wywołane exit w programie potomnym program macierzysty wznawia pracę
- Program macierzysty przechwytuje wartość zwracaną przez wait i informuje o statusie zakończenia procesu potomnego

Procesy zombi

- Kiedy proces potomny kończy pracę, jego powiązanie z procesem macierzystym jest podtrzymywane, dopóki ten nie zakończy działania albo nie wykona wait
- Proces potomny nie jest już aktywny, ale pozostaje po nim wpis w tablicach opisu procesów



Wprowadzenie procesu zombie – przykładowa aplikacja

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{       pid_t pid; int n;
        char *message;
        printf("fork program starting\n");
        pid = fork();
        switch(pid)
        {       case -1: perror("fork failed");
                         exit(1);
                case 0:          message = "This is the child";
                                 n = 3;
                                 break;
                default: message = "This is the parent";
                                 n = 5;
                                 break;
}       for(; n > 0; n--) {puts(message);
        sleep(1);
        }
        exit(0);}
```

```
F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD
004 S 0 1273 1259 0 75 0 - 589 wait4 pts/2 00:00:00 su
000 S 500 1465 1262 0 75 0 - 2569 schedu pts/1 00:00:01 emacs
000 S 500 1603 1262 0 75 0 - 313 schedu pts/1 00:00:00 fork2
003 Z 500 1604 1603 0 75 0 - 0 do_exi pts/1 00:00:00 fork2 <defunct>
000 R 500 1605 1262 0 81 0 - 781 - pts/1 00:00:00 ps
```

Gdy proces macierzysty zakończy swoją pracę nieprawidłowo, proces potomny otrzyma proces macierzysty z identyfikatorem 1 (init).

Oczekiwanie na określony proces

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

Wywołanie funkcji waitpid z parametrami:

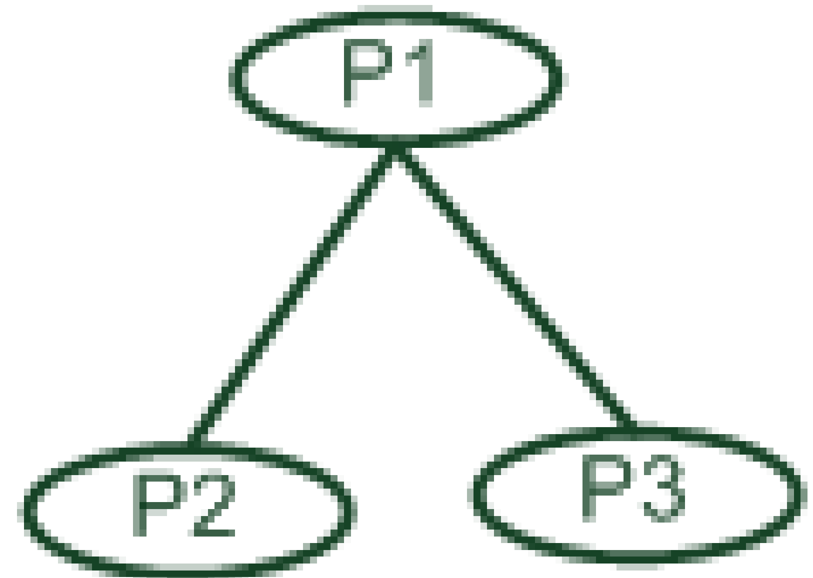
```
waitpid(child_pid, (int *) 0, WNOHANG);
```

pozwala na monitorowanie stanu procesu potomnego.
Funkcja zwróci:

- 0 – jeśli proces potomny nie zakończył pracy
- PID potomka, jeśli proces potomny zakończy pracę normalnie
- -1 i ustawi errno, gdy zostanie wykryty błąd

Użycie fork do utworzenia 2 procesów potomnych

```
Kod wspólny
if(fork() == 0) {
   Kod procesu potomnego P2
   exit(0);
}
if(fork() == 0) {
   Kod procesu potomnego P3
   exit(0);
}
Kod procesu macierzystego P1
```



Użycie fork do tworzenia kaskady procesów potomnych

```
Kod wspólny
if(fork() == 0) {
  if(fork() == 0 {
    Kod procesu potomnego P3
    exit(0);
  } else {
    Kod procesu potomnego P2
    exit(0);
  }
 }

 Kod procesu macierzystego P1

 exit(0);
```

Pliki

- W Linuksie wszystko jest plikiem (prawie)
- Dostęp do większości urządzeń (plików dyskowych, drukarek, konsoli, portów szeregowych i wielu innych) jest taki sam i odbywa się jak dostęp do plików
- Nieco inaczej odbywa się dostęp do sieci (gniazda)
- W dostępie do wymienionych urządzeń można się posłużyć zbiorem tzw. niskopoziomowych funkcji dostępu do plików: open, close, read, write i ioctl.

Specjalne pliki

• /dev/console	domyślna konsola
• /dev/tty	terminal
• /dev/null	puste urządzenie

Podstawowe funkcje niskiego poziomu do obsługi plików i urządzeń

- open: otwórz plik lub urządzenie
- read: czytaj z otwartego pliku lub urządzenia
- write: zapisz coś do otwartego urządzenia lub pliku
- close: zamknij plik lub urządzenie
- ioctl: przekaż informacje sterujące do sterownika urządzenia

Domyślne deskryptory plików

- Każdy uruchomiony program ma powiązane ze sobą deskryptory plików.
- Domyślnie są to:
  - 0: standardowe wejście
  - 1: standardowe wyjście
  - 2: standardowe błędy

write()

```
#include <unistd.h>
size_t write(int fildes, const void *buf, size_t nbytes);
```

- fildes – deskryptor pliku (uzyskany z open)
- buf – bufor z danymi do zapisu
- nbytes – ilość danych do zapisu
- Funkcja zwraca ilość zapisanych bajtów, 0 gdy koniec pliku, -1 gdy błąd

Przykład:

```
#include <unistd.h>
#include <stdlib.h>
int main()
{ if ((write(1, “Here is some data\n”, 18)) != 18)
  write(2, “A write error has occurred on file descriptor 1\n”,46);
  exit(0);
}
```

```
$ simple_write
Here is some data
$
```

read()

```
#include <unistd.h>
size_t read(int fildes, void *buf, size_t nbytes);
```

- fildes – deskryptor pliku
- buf – bufor na dane
- nbytes – maksymalna ilość bajtów do odczytania
- Funkcja zwraca ile bajtów odczytała, 0-nic nie przeczytano koniec pliku, -1 – błąd.

```
#include <unistd.h>
#include <stdlib.h>
int main()
{  char buffer[128];
   int nread;
   nread = read(0, buffer, 128);
   if (nread == -1)
      write(2, “A read error has occurred\n”, 26);
   if ((write(1,buffer,nread)) != nread)
      write(2, “A write error has occurred\n”,27);
   exit(0);
}
```

Przekierowanie strumienia wyjściowego jednego polecenia na drugie

```
$ echo hello there | simple_read
hello there
$ simple_read < draft1.txt
Files
In this chapter we will be looking at files and
directories and how to manipulate
them. We will learn how to create files, o$
```

open()

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
int open(const char *path, int oflags);
int open(const char *path, int oflags, mode_t mode);
```

- path – ścieżka dostępu do pliku

oflags:

Tryb	Opis
O_RDONLY	Otwórz tylko do odczytu
O_WRONLY	Otwórz tylko do zapisu
O_RDWR	Otwórz do zapisu i odczytu
O_APPEND	Ustaw się na końcu pliku
O_CREAT	Utwórz plik jeśli potrzeba
O_EXCL	W połączeniu z O_CREAT zapewnia, że tylko wywołujący utworzy plik

mode:

- S_IRUSR: prawo czytania, właściciel
- S_IWUSR: prawo pisania, właściciel
- S_IXUSR: prawo wykonywania, właściciel
- S_IRGRP: prawo czytania, grupa
- S_IWGRP: prawo pisania, grupa
- S_IXGRP: prawo wykonywania, grupa
- S_IROTH: prawo czytania, inni
- S_IWOTH: prawo pisania, inni
- S_IXOTH: prawo wykonywania, inni

przykład:

```
open (“myfile”, O_CREAT, S_IRUSR|S_IXOTH);
```

close()/ ioctl()

```
#include <unistd.h>
int close(int fildes);
```

```
#include <unistd.h>
int ioctl(int fildes, int cmd, ...);
```

Wykonaj komendę cmd na pliku wskazywanym Przez fildes uwzględniając ew. dodatkowe pa-rametry.

Przykład zastosowania

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
int main()
{
char c;
int in, out;
in = open(“file.in”, O_RDONLY);
out = open(“file.out”, O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
while(read(in,&c,1) == 1)
		write(out,&c,1);
exit(0);
}
```

Standardowa biblioteka wejścia-wyjścia - przypomnienie

- fopen, fclose
- fread, fwrite
- fflush
- fseek
- fgetc, getc, getchar
- fputc, putc, putchar
- fgets, gets
- printf, fprintf, and sprintf
- scanf, fscanf, and sscanf

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
int c;
FILE *in, *out;
in = fopen(“file.in”,”r”);
out = fopen(“file.out”,”w”);
while((c = fgetc(in)) != EOF)
fputc(c,out);
exit(0);
}
```

Definicja potoku

- Terminu potok (pipe) używa się na określenie przepływu danych z jednego procesu do innego. Można powiedzieć, że potoki łączą wyjście jednego procesu z wejściem innego.
- Istnieje możliwość wywołania polecenia powłoki, które wiąże:
  - Standardowe wejście polecenia2 z klawiaturą terminala
  - Standardowe wyjście polecenia2 ze standardowym wejściem polecenia1
  - Standardowe wyjście polecenie1 z ekranem komputera:

```
polecenie2 | polecenie1
```

Przykład do przetestowania potoków ustalonych z linii poleceń

```c
// polecenie2
#include <stdio.h>
#include <stdlib.h>

int main()
{
   printf("Hello world!\n");
   return 0;
}
```

```c
// polecenie1
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main()
{
   char txt[100];
   int i=0;
   gets(txt);
   while(txt[i]!=0)
   {
      txt[i]=toupper(txt[i]);
      i++;
   }
   puts(txt);
   return 0;
}
```

Funkcja pipe (potok nie nazwany)

```
#include <unistd.h>
int pipe(int file_descriptor[2]);
```

- Udostępnia ona taki mechanizm przekazywania danych pomiędzy dwoma programami, który nie wymaga uruchomienia powłoki w celu interpretacji żądanego polecenia
- Do pipe przekazywana jest tablica (wskaźnik do tablicy) dwóch liczb całkowitych będących deskryptorami plików.
- Funkcja wypełnia tablicę dwoma nowymi deskryptorami plików i zwraca zero w przypadku błędy zwraca -1 i ustawia zmienną errno.
- Dwa zwrócone deskryptory plików są połączone w specjalny sposób. Wszystkie dane zapisane do file_descriptor[1], mogą być odczytane przez file_descriptor[0]. Dane są przetwarzane według zasady FIFO
- Przesyłanie danych z i do takiego potoku musi się odbywać z zastosowaniem funkcji read, write a nie fread, fwrite.

Pierwsze zastosowanie funkcji pipe

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{ int data_processed;
  int file_pipes[2];
  const char some_data[] = “123”;
  char buffer[BUFSIZ + 1];
  memset(buffer, ‘\0’, sizeof(buffer));
  if (pipe(file_pipes) == 0)
  { data_processed = write(file_pipes[1], some_data, strlen(some_data));
    printf(“Wrote %d bytes\n”, data_processed);
    data_processed = read(file_pipes[0], buffer, BUFSIZ);
    printf(“Read %d bytes: %s\n”, data_processed, buffer);
    exit(EXIT_SUCCESS);
   }
   exit(EXIT_FAILURE);
}
```

```
$ ./pipe1
Wrote 3 bytes
Read 3 bytes: 123
```

Potoki i fork

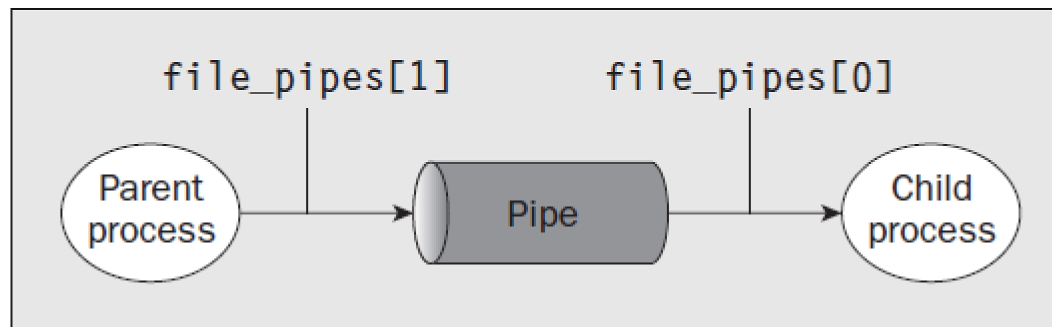
```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{ int data_processed;
  int file_pipes[2];
  const char some_data[] = “123”;
  char buffer[BUFSIZ + 1];
  pid_t fork_result;
  memset(buffer, ‘\0’, sizeof(buffer));
  if (pipe(file_pipes) == 0)
  { fork_result = fork();
    if (fork_result == -1)
    { fprintf(stderr, “Fork failure”);
      exit(EXIT_FAILURE);
    }
```

```
if (fork_result == 0)
{ data_processed = read(file_pipes[0], buffer,
                                   BUFSIZ);
   printf(“Read %d bytes: %s\n”, data_processed,
             buffer);
   exit(EXIT_SUCCESS);
}
else
{ data_processed = write(file_pipes[1],
                           some_data,strlen(some_data));
   printf(“Wrote %d bytes\n”, data_processed);
}
}
exit(EXIT_SUCCESS);
}
```

```
$ ./pipe2
Wrote 3 bytes
Read 3 bytes: 123
```

Jak to działa?

- Najpierw program tworzy potok, używając funkcji pipe
- Po sprawdzeniu, że funkcja fork zakończyła się pomyślnie proces macierzysty zapisuje dane do potoku, a proces potomny je odczytuje
- Proces macierzysty i potomny kończą pracę po pojedynczym wykonaniu instrukcji write/read
- Proces potomny dziedziczy w momencie wykonywania funkcji fork utworzone deskryptory plików
- Uzyskano możliwość przekazywania danych pomiędzy procesami.



Czytanie zamkniętych potoków

- Zwykle dane z potoków odczytuje się porcjami w pętli
- Funkcja read blokuje proces – oczekuje aż będą dostępne następne dane
- Może się okazać, że potok od strony nadawcy zostaje zamknięty
- Wtedy funkcja read zwraca 0 (-1, gdy wykryto błąd otwarcia pliku) i program może zapobiec blokowaniu
- Jeśli korzystamy z potoku utworzonego dla dwu procesów z zastosowaniem funkcji fork, to w programie istnieją 2 deskryptory plików do zapisu w potoku.
- Abu potok uznał, że jest zamknięty oba z nich muszą zostać zamknięte.

Zamykanie nienazwanych potoków - przykład

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int
main(int argc, char *argv[])
{
    int pipefd[2];
    pid_t cpid;
    char buf;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <string>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    cpid = fork();
    if (cpid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
```

```
    if (cpid == 0) {            /* Child reads from pipe */
        close(pipefd[1]);       /* Close unused write end */

        while (read(pipefd[0], &buf, 1) > 0)
            write(STDOUT_FILENO, &buf, 1);

        write(STDOUT_FILENO, "\n", 1);
        close(pipefd[0]);
        _exit(EXIT_SUCCESS);

    } else {                    /* Parent writes argv[1] to pipe */
        close(pipefd[0]);       /* Close unused read end */
        write(pipefd[1], argv[1], strlen(argv[1]));
        close(pipefd[1]);       /* Reader will see EOF */
        wait(NULL);             /* Wait for child */
        exit(EXIT_SUCCESS);
    }
}
```

Nazwane potoki: FIFO

- Umożliwiają komunikację pomiędzy procesami, które nie są ze sobą spokrewnione
- Są specjalnym rodzajem pliku, który istnieje w systemie plików, ale zachowuje się jak nie nazwane potoki, które były omawiane do tej pory
- Polecenie systemowe tworzące nazwany potok: **mkfifo filename**
- Z poziomu programu można utworzyć nazwany potok z zastosowaniem funkcji:

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *filename, mode_t mode);
int mknod(const char *filename, mode_t mode | S_IFIFO, (dev_t) 0);
```

Tworzenie nazwanego potoku

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
int main()
{  int res = mkfifo("/tmp/my_fifo", 0777);
   if (res == 0) printf("FIFO created\n");
   exit(EXIT_SUCCESS);
}
```

```
ubuntu@ubuntu:~$ mkfifo /tmp/fifo1
ubuntu@ubuntu:~$ cat < /tmp/fifo1 &
[1] 13337
ubuntu@ubuntu:~$ echo "Ala ma kota" > /tmp/fifo1
ubuntu@ubuntu:~$ Ala ma kota

[1]+ Done cat                    < /tmp/fifo1
```

Otwieranie FIFO funkcją open

- open(const char *path, O_RDONLY);
  Open się zablokuje, to znaczy nie powróci, dopóki inny proces nie otworzy tego samego potoku do zapisu
- open(const char *path, O_RDONLY | O_NONBLOCK);
  Funkcja open zakończy się teraz pomyślnie i natychmiast powróci, nawet jeśli żaden proces nie otworzy FIFO do zapisu
- open(const char *path, O_WRONLY);
  W tym przypadku funkcja open zablokuje się, dopóki inny proces nie otworzy tego samego FIFO do odczytu
- open(const char *path, O_WRONLY | O_NONBLOCK);
  To wywołanie zawsze natychmiast wraca, ale jeśli wcześniej żaden proces nie otworzył FOFO do odczytu, open zwróci -1 i FIFO nie zostanie otwarte.
- Funkcja open może zostać zastosowana do **synchronizacji** pracy procesów.

FIFO: odczyt i zapis

- Użycie trybu O_NONBLOCK ma wpływ na zachowanie funkcji write i read.
  - Odczyt (read) z pustego, blokującego się FIFO (otwartego bez znacznika O_NONBLOCK) będzie oczekiwał, aż będzie można odczytać jakieś dane, natomiast odczyt z nieblokującego się FIFO, w którym nie ma żadnych danych, zwróci 0.
  - Zapis (write) do pełnego, blokującego się FIFO będzie oczekiwał, aż można będzie zapisać dane. Zapis do FIFO, które nie może przyjąć wszystkich bajtów, może zadziałać na 2 sposoby:
    - Spowodować błąd, ponieważ nie można przesłać wszystkich danych
    - Zapisać część danych, zwracając liczbę rzeczywiście wysłanych danych

Komunikacja międzyprocesowa przy użyciu FIFO – producent (fifo3)

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF
#define TEN_MEG (1024 * 1024 * 10)
int main()
{ int pipe_fd; int res;
  int open_mode = O_WRONLY;
  int bytes_sent = 0;
  char buffer[BUFFER_SIZE + 1];
  if (access(FIFO_NAME, F_OK) == -1)
  { res = mkfifo(FIFO_NAME, 0777);
    if (res != 0)
    {  fprintf(stderr, "Could not create fifo %s\n",
                                        FIFO_NAME);
       exit(EXIT_FAILURE);
    }
  }
```

```
  printf("Process %d opening FIFO O_WRONLY\n",
                                        getpid());
  pipe_fd = open(FIFO_NAME, open_mode);
  printf("Process %d result %d\n", getpid(),
                                        pipe_fd);
  if (pipe_fd != -1)
  {  while(bytes_sent < TEN_MEG)
    {   res = write(pipe_fd, buffer, BUFFER_SIZE);
        if (res == -1)
       {  fprintf(stderr, "Write error on pipe\n");
          exit(EXIT_FAILURE);
       }
       bytes_sent += res;
    }
    (void)close(pipe_fd);
  }
  else
  {  exit(EXIT_FAILURE);
  }
  printf("Process %d finished\n", getpid());
  exit(EXIT_SUCCESS);
}
```

Komunikacja międzyprocesowa przy użyciu FIFO – konsument (fifo4)

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF
int main()
{   int pipe_fd; int res;
    int open_mode = O_RDONLY;
    char buffer[BUFFER_SIZE + 1];
    int bytes_read = 0;
    memset(buffer, '\0', sizeof(buffer));
    printf("Process %d opening FIFO
    O_RDONLY\n", getpid());
```

```
    pipe_fd = open(FIFO_NAME,
                         open_mode);
    printf("Process %d result %d\n",
                        getpid(), pipe_fd);
    if (pipe_fd != -1)
    {  do
       {   res = read(pipe_fd, buffer,
                            BUFFER_SIZE);
           bytes_read += res;
       } while (res > 0);
       (void)close(pipe_fd);
    }
    else
    {   exit(EXIT_FAILURE);
    }
    printf("Process %d finished, %d bytes
    read\n", getpid(), bytes_read);
    exit(EXIT_SUCCESS);
}
```

Przykładowy rezultat pracy podanych 2 programów

```
$ ./fifo3 &
[1] 375
Process 375 opening FIFO O_WRONLY
$ time ./fifo4
Process 377 opening FIFO O_RDONLY
Process 375 result 3
Process 377 result 3
Process 375 finished
Process 377 finished, 10485760 bytes read

real    0m0.053s
user    0m0.020s
sys     0m0.040s

[1]+ Done fifo3
```

- Oba programy korzystają z FIFO w trybie blokującym
- Najpierw uruchamiany jest producent, który blokuje się, czekając na otwarcie FIFO przez program odczytujący
- Kiedy uruchomiony zostanie konsument, program zapisujący odblokowuje się i zaczyna wprowadzać dane do potoku
- W tym samym czasie program odczytujący rozpoczyna czytanie danych z potoku